# Module-Wise Compilation for a Language with Type-Parameterization Mechanism (Mathematical Methods in Software Science and Engineering)

AUTHOR(S):

YUASA, TAIICHI

CITATION:

YUASA, TAIICHI. Module-Wise Compilation for a Language with Type-Parameterization Mechanism (Mathematical Methods in Software Science and Engineering). 数理解析研究所講究録 1979, 363: 1-40

ISSUE DATE:

1979-09

URL:

http://hdl.handle.net/2433/104571

数理解析研究所講究録
363巻 1979年 1-40



RIMS-280

# Module-wise Compilation for a Language with Type-parameterization Mechanism

By

Taiichi Yuasa

Research Institute for Mathematical Sciences

Kyoto University, Kyoto, Japan

May 1979

abstract

Language ℓ is a specification and programming language designed to support hierarchical and modular program development. The notion of *sypes*, which generalizes the so-called type-parameterization mechanism, causes some essential problems in the implementation of the language. These problems are discussed in detail and what is considered to be an efficient technique is introduced with which each type-parameterized module is separately compiled independent of the context in which it is used with actual type parameters.

## 1. Introduction

Structured programming with data and procedural abstraction mechanisms has been shown to highly increase program readability and software reliability. (See, e.g., CLU [2]) It has also been shown that the difficulties of specification and verification of programs, especially for large scale software systems, can be eased by introducing hierarchical structures into programs. Language ℓ [4] has been proposed to support such program development with hierarchical and modular structures.

One of the characteristics of ℓ is that it has a new data type concept --sypes, which generalizes the so-called type-parameterization mechanism [5].

### Sample programs with comments

We list some programs written in language ℓ in order to locate the problem to be discussed in this paper. As complete description of the language is not within the scope of this paper, we only give preliminary remarks along with the programs. For a detailed explanation of the language or formal definitions for, say, the sype-sype relation, refer to [5].

```
interface type INT

+     fn  ZERO: --------> @      as 0
+         ONE:  --------> @      as 1
+         ADD: (@,@) ---> @      as @+@
+         MULT:(@,@) ---> @      as @*@
+         REV:  @ -------> @      as -@
          GE:  (@,@) ---> @      as @≤@
                .
                .

end interface

specification type INT

      var X,Y,Z:@
+     axiom   1:  X+0=X
+             2:  X+Y=Y+X
+             3:  (X+Y)+Z=X+(Y+Z)
+             4:  X*Y=Y*X
+             5:  X*(Y*Z)=(X*Y)*Z
+             6:  X*(Y+Z)=X*Y+X*Z
+             7:  1*X=X
+             8:  X+(-X)=0
              9:  X≤Y ∨ Y≤X
             10:  (X≤Y&Y≤Z)==>X≤Z
             11:  (X≤Y&Y≤X)==>X=Y
                    .
                    .
end specification
```

This is part of the type module INT which presents the type of integers. The interface part declares the primitive funtions on INT with their domains and ranges. @ denotes the type presented by the type module, which is INT in this case. By as a notational abbreviation is introduced for a function name. For instance, ADD(X,Y) can be written as X+Y. In the specification part, the basic axioms on INT are placed.

The following is a type module which presents RAT or the type of rational numbers.

```
interface type RAT

+     fn  ZERO: --------> @      as 0
+         ONE:  --------> @      as 1
+         ADD: (@,@) ---> @      as @+@
+         MULT:(@,@) ---> @      as @*@
+         REV:  @ ------> @      as -@
          INV:  @ ------> @      as /@
                .
                .

end interface

specification type RAT

      var X,Y,Z:@
+     axiom   1:  X+0=X
+             2:  X+Y=Y+X
+             3:  (X+Y)+Z=X+(Y+Z)
+             4:  X*Y=Y*X
+             5:  X*(Y*Z)=(X*Y)*Z
+             6:  X*(Y+Z)=X*Y+X*Z
+             7:  1*X=X
+             8:  X+(-X)=0
              9:  X≠0 ==> X*(/X)=1
                    .
                    .
end specification
```

Notice that these two types have a common substructure: They have five primitive functions in common and their basic axioms from 1 to 8 are identical. Since this substructure may be contained in many other types, we extract and isolate the lines preceded by + to form the type module of RING.

```
interface sype RING

     fn  ZERO: --------> @      as 0
         ONE:  --------> @      as 1
         ADD: (@,@) ---> @      as @+@
         MULT:(@,@) ---> @      as @*@
         REV:  @ -------> @     as -@

end interface
```

```
specification sype RING

     var X,Y,Z:@
     axiom   1:  X+0=X
             2:  X+Y=Y+X
             3:  (X+Y)+Z=X+(Y+Z)
             4:  X*Y=Y*X
             5:  X*(Y*Z)=(X*Y)*Z
             6:  X*(Y+Z)=X*Y+X*Z
             7:  1*X=X
             8:  X+(-X)=0

end specification
```

We introduce a sype-type relation "$\lesssim$". For a sype S and a type T, $S \lesssim T$ holds if T contains S as its substructure. Thus $RING \lesssim INT$ and $RING \lesssim RAT$ in this case. To establish $S \lesssim T$, for each primitive function of S, say f, there must be defined a function of T of the same name (i.e. T#f). T#f is said to be the function corresponding to f of sype S.

In a similar way, we construct the sype module FIELD. Here we have FIELD $\lesssim$ RAT.

```
specification sype FIELD

     fn  ONE:  --------> @      as 1
         ZERO: --------> @      as 0
         MULT:(@,@) ---> @      as @*@
         ADD: (@,@) ---> @      as @+@
         INV:  @ ------> @      as /@
         REV:  @ ------> @      as -@

end interface

specification sype FIELD

     var X,Y,Z:@
     axiom   1:  X+0=X
             2:  X+Y=Y+X
             3:  (X+Y)+Z=X+(Y+Z)
             4:  X*Y=Y*X
             5:  X*(Y*Z)=(X*Y)*Z
             6:  X*(Y+Z)=X*Y+X*Z
             7:  1*X=X
             8:  X+(-X)=0
             9:  X ≠ 0 ==> X*(/X)=1

end specification
```

Now we define a type module POLY(P:RING) which presents the type of polynomials in one variable with any coefficient type T such that RING $\lesssim$ T.

```
interface type POLY(P:RING)

     fn  ZERO: --------> @      as 0
         ONE:  --------> @      as 1
         ADD: (@,@) ---> @      as @+@
         MULT:(@,@) ---> @      as @*@
         REV:  @ ------> @      as -@
         COEF:(@,INT)--> @
         DEG:  @ ------> INT

end interface


specification type POLY(P:RING)

     var X,Y,Z:@
     axiom   1:  X+0=X
             2:  X+Y=Y+X
             3:  (X+Y)+Z=X+(Y+Z)
             4:  X*Y=Y*X
             5:  X*(Y*Z)=(X*Y)*Z
             6:  X*(Y+Z)=X*Y+X*Z
             7:  1*X=X
             8:  X+(-X)=0
                  .
                  .
end specification


realization type POLY(P:RING)

     rep=ARRAY(P)
           .
           .
     fn  ↓ADD(X,Y:rep) return (Z:rep)

          var  I:INT
                    .
                    .
               Z[I] := P#ADD(X[I],Y[I]) ............(*)
                    .
                    .
     end fn
           .
           .
end realization
```

Fig.1.1 Type module POLY(P:RING)

An arbitrary type T such that RING$\lesssim$T can be used as the actual type parameter for POLY(P:RING). For instance, since RING$\lesssim$INT, POLY(INT) is a type of polynomials whose coefficients are of type INT. Thus P, which we call a type parameter of type RING, represents the indefinite (formal) type parameter and POLY(P:RING) is said to be a type-parameterized module. We call POLY(INT) a definite module-instance of POLY(P:RING) since the actual type parameter INT is a definite type. On the other hand, ARRAY(P) in the realization part of POLY(P:RING) or POLY(POLY(P1)) which will appear later in Fig.1.2. are called indefinite module instances, for they contain formal type parameters P of POLY(P:RING) or P1 of BIPOLY(P1:RING).

The realization part gives an implementation of POLY(P:RING). Each object of type POLY(P:RING) is represented by ARRAY(P) or array of type P. (e.g. POLY(INT) is represented by ARRAY(INT).) There is a rigorous distinction in the language between an abstract function (which is presented in the interface and the specification part) and its concrete function (which defines an implementation of the corresponding abstract function). To discriminate between these two kinds of functions, each concrete function has the name of its corresponding abstract function preceded by "↓". In the figure above, the concrete function corresponding to (abstract) ADD has the name ↓ADD.

The line marked "*" says that the I-th components of X

and Y are 'added' and then the result replaces the I-th component of Z. Since the components of X and Y are of type P, the addition + must be that of P (i.e. P#ADD). In this paper, those functions which are actually executed in runtime at the line "*" are said to be actual ADD's for P#ADD. If the actual type parameter is INT, the actual ADD is the addition of integers, i.e. INT#ADD.

From the interface and specification parts of POLY(P:RING), we find another sype-type relation RING$\lesssim$POLY(P:RING). Remember that any type T such that RING$\lesssim$T can be used as the actual type parameter for POLY(P:RING). This indicates that POLY(POLY(P:RING)) is permissible. Indeed, a type module BIPOLY(P1:RING) is represented by POLY(POLY(P:RING)), which is supposed to present the type of polynomials in two variables.

```
realization type BIPOLY(P1:RING)
     rep=POLY(POLY(P1))
          .
          .
     fn  ↓ADD(X,Y:rep) return (Z:rep)
                .
                .
           Z := rep#ADD(X,Y)
                .
                .
     end fn
          .
          .
end realization
```

Fig.1.2 Type module BIPOLY(P1:RING)

(Note that POLY(POLY(P1))#ADD is abbreviated as rep#ADD.)

The relation "$\lesssim$" is also defined between two sypes in language $\iota$. For example, FIELD has RING as its substructure. Thus we can denote RING$\lesssim$FIELD.

```
realization procedure STP(P2:FIELD)
     var A,B,C:POLY(P2)
          .
          .
     fn  ↓F....
             .
             .
         C := POLY(P2)#ADD(A,B)
             .
             .
     end fn
          .
          .
end realization
```

Fig.1.3 Procedure module STP(P2:FIELD)

In the body of STP(P2:FIELD)#F, above, POLY(P2)#ADD is called. That is, the actual type parameter which STP(P2:FIELD) receives in execution time is passed to POLY(P:RING). This is permissible since RING$\lesssim$FIELD.

We have already used such a sype=sype relation in the realization part of POLY(P:RING). The built-in module ARRAY(P3:ANY) is a type-parameterized module which receives a type paremeter P3 of sype ANY. Sype ANY is a built-in sype whose only primitive function is EQUAL or equality.

```
interface sype ANY

     fn EQUAL:  (@,@) --> BOOL    as @=@

end interface

specification sype ANY

     var X,Y,U,V:@
     axiom  1:  X=X
            2:  (X=Y&U=V) ==> (X=U)=(Y=V)

end specification
```

In language ℓ , every sype or type is supposed to have its own EQUAL function. It can be defined explicitly in the module or else it is automatically defined by the system. Thus any sype or type S satisfies ANY≲S. Since ANY RING holds, ARRAY(P) is permissible in the realization part of POLY(P:RING).

Although the type-parameterization mechanism itself is found in some other languages (e.g. CLU[2]), the expressive power of the notion of sypes brings some new difficulties into the implementation of the language.

This paper discusses these difficulties and shows how to overcome them. Section 2 presents the most straightforward way of compiling type-parameterized modules, called the 'definite module-instance approach'. Since the method has some deficiencies, we would prefer another method with which each type-parameterized module is separately compiled independent of the context in which it is used with actual type parameters. Then, in section 3, we discuss what kind of information is required for the actual type

parameters. Finally, in section 4, we explain how such information is constructed in execution time.

## The problem

Let us return to the module POLY(P:RING) (in Fig.1.1) and focus on the following problem: what should the compiler do in processing the realization part of POLY(P:RING), especially for the function call of P#ADD (marked "*")? Also what kind of information should be sent to POLY(P:RING) in execution time?

2. A solution -- definite module-instance approach

One possible solution is to do almost nothing with POLY(P:RING) itself until P is bound to some actual type parameter. When POLY(T) is used in other modules (i.e. when P is bound to an actual definite type instance T), the instance of the realization part of POLY(P:RING), with all occurences of P replaced by T, is processed.

For example, when POLY(INT) is used, the line marked "*" is replaced by:

```
Z[I] := INT#ADD(X[I],Y[I])
```

Then the processor knows that INT#ADD is to be called.

If POLY(P:RING) is used with the actual type parameter RAT or the type of rational numbers, we have another definite module-instance POLY(RAT) with:

```
Z[I] := RAT#ADD(X[I],Y[I])
```

Thus the processor actually regards these module-instances of POLY(P:RING) as two different type modules. Notice that the number of module-instances of a module is always finite because any module in language ℓ must be hierarchical i.e. no module can depend on itself. (Refer to [5]. The proof of finiteness is found in [8].) Therefore this approach is valid. Indeed, the experimental version of the ℓ-language compiler adopted this method.

This is not altogether a bad solution. Without the type-parameterization mechanism, one must define, say, two non-type-parameterized modules INTPOLY and RATPOLY separately, corresponding to the module instances POLY(INT)

and POLY(RAT), respectively. Here, INTPOLY and RATPOLY are thought to be of completely different modules. Thus the above method is nothing more than the conventional way of processing modules without the type-parameterization mechanism. In addition, the above method makes some optimizations possible. For example, as INT#ADD in POLY(INT) is nothing more than the usual integer addition, one can generate a single machine instruction instead of the actual function call of INT#ADD.

However, this method has the following deficiencies.

1. The bookkeeping of all instances of all type-parameterized modules is not a trivial task and is also time-consuming. (See, for example, in Fig.1.3 when STP(RAT) is defined, POLY(RAT) is to be automatically and implicitly defined.)
2. The compilation time tends to be long with repetitions of similar processing. Besides, a large amount of storage is required since each instance of a single type-parameterized module must be allocated separately.
3. Since a type-parameterized module is defined independently of the actual type parameters it receives, it is often convenient in program development to process it independently. For example, a type parameter independent object code of a type-parameterized module may make it possible to debug the module without sending actual type parameters.

Thus we would rather have module-wise processing where

each module is independently compiled and type parameter bindings are done dynamically. The following sections are devoted to showing how this can be done.

3. What is sent as actual type parameters?

Procedure tables for sype-type relations

Given a procedure module AHO, in the realization part of which POLY(INT)#ADD is called,

```
realization procedure AHO
     .
     .
     POLY(INT)#ADD(X,Y)
     .
     .
end realization
```

let us consider what kind of information AHO must send to (the compiled) POLY(P:RING) (in addition to the usual parameter information for X and Y).

The actual ADD for P#ADD in the realization of POLY(P:RING)#ADD is INT#ADD in this case. Thus the information must include the location of INT#ADD. Since AHO does not know the realization part of POLY(P:RING), AHO cannot determine which functions, corresponding to the functions primitive on sype RING, are actually used in POLY(P:RING)#ADD. Therefore AHO must send a table which contains all actual functions corresponding to the primitive functions of sype RING. We call such a table procedure table for RING$\lesssim$INT and denote it as PT<RING,INT>.

In general, for each pair of type S and type T such that S$\lesssim$T and T is used as an actual type parameter of sype S, PT<S,T> is constructed as follows. Let f1,...,fn be the primitive functions which are defined in that order in the interface part of sype S and let T#f1,...,T#fn be the

functions of type T corresponding to f1,...,fn, respectively. PT<S,T> is a block of n entries and its i-th entry (1≤i≤n) contains the entry point of the function T#fi.

For instance, since the third function declared in sype RING is ADD and fourth one is MULT, the third and fourth entries of PT<RING,INT> contain the location of INT#ADD and INT#MULT, respectively.

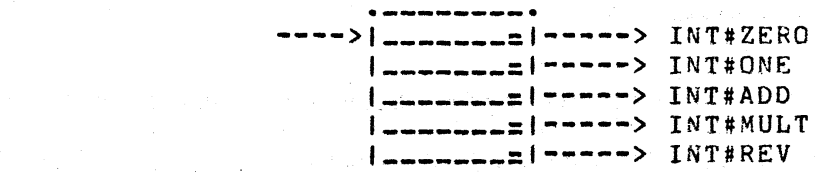



Fig.3.1 Procedure table PT<RING,INT>

At the time of compilation of POLY(P:RING), the processor recognized that ADD is the third function of sype RING by analyzing the interface part of RING. The object code is made so that the third entry of the procedure table is used in order to access the actual ADD. Then, when POLY(INT)#ADD is called in AHO, the location of PT<RING,INT> is sent to POLY(P:RING).

Note that the order of primitive functions in the interface part of sype RING is important and needs to be fixed once POLY(P:RING) is compiled.

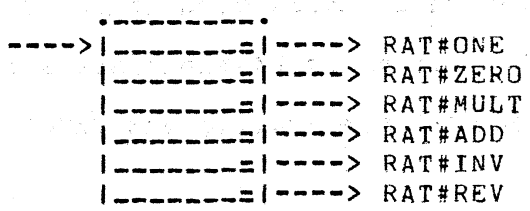
## Adaptor tables for sype-sype relations

Suppose we have a procedure module MAKO, in the

realization part of which STP(RAT)#F is called. (See Fig.1.3)

```
realization procedure MAKO
     .
     .
     STP(RAT)#F
     .
     .
end realization
```

As explained before, PT<FIELD,RAT> is sent to STP(P2:FIELD) when STP(RAT)#F is called in executing (a body in the realization part of) MAKO.

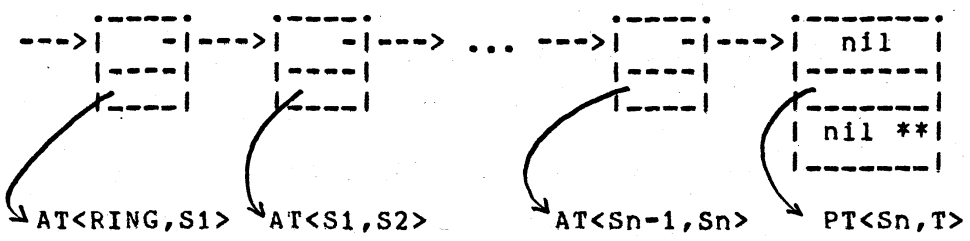



In STP(P2:FIELD), however, this procedure table cannot be directly sent to POLY(P:RING) for the following reason: POLY(P:RING) expects a procedure table in which the functions are ordered according to the interface part of the sype RING, but the order of the primitive functions in RING does not necessarily coincide with that of the corresponding primitive functions of FIELD. Indeed, the location of RAT#ADD is found in the fourth entry in PT<FIELD,RAT> while ADD is the third function in the interface part of sype

RING.

Thus some adaptations must be made to use PT<FIELD,RAT> in POLY(P:RING). To this end, we introduce another kind of table called adaptor tables. For each pair of sypes S and S' such that $S \lesssim S'$, an adaptor table AT<S,S'> is constructed. If the i-th primitive function of S is presented as the j-th primitive function in the interface part of S', then the i-th entry of AT<S,S'> has the value of j. (Actually, however, AT<S,S'> is not required if the order of the primitive functions in S coincides with that of the corresponding functions of S'.)

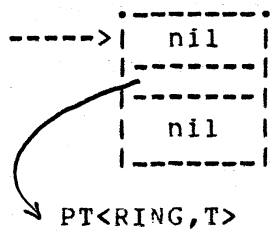
POLY(P:RING) is supposed to receive a single list called the procedure description list (PDL) of the following form, where S1,...,Sn are distinct sypes and T is a type such that $RING \lesssim S1$, $S1 \lesssim S2$,..., and $Sn \lesssim T$.



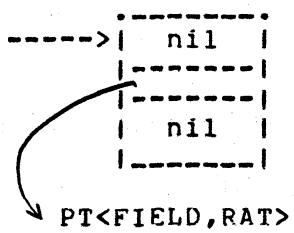

As a particular (but most common) case, n may be zero. That is, the PDL is simply of the form:

---

**) This cell is used for the 'type parameter list' explained later.




In the above example, when MAKO is compiled, the processor constructs the following ;




In the compilation time of STP(P1:FIELD), an incomplete PDL shown below is prepared with AT<RING,FIELD>. (It is incomplete in the sense that the cell marked "*" must be linked to form a PDL in execution time.)

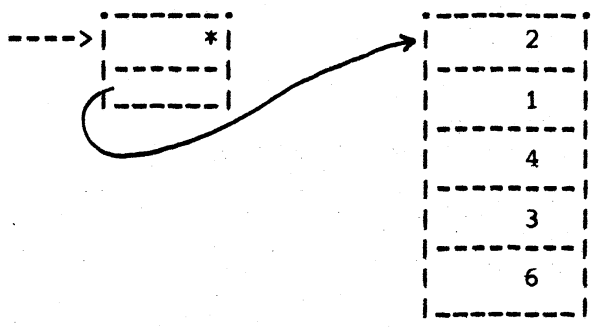



Fig.3.2 AT<RING,FIELD>

In execution time, this incomplete PDL is linked to the PDL

that STP(P1:FIELD) receives and is sent to POLY(P:RING) when POLY(P1) ADD is called in executing STP(P1:FIELD).

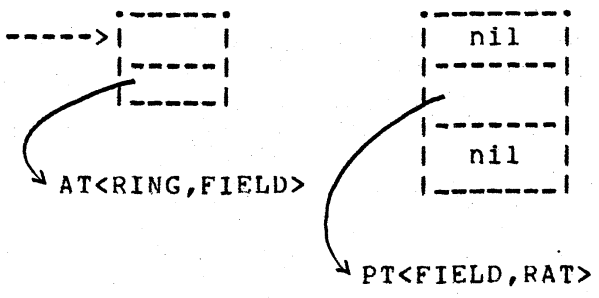



Fig.3.3

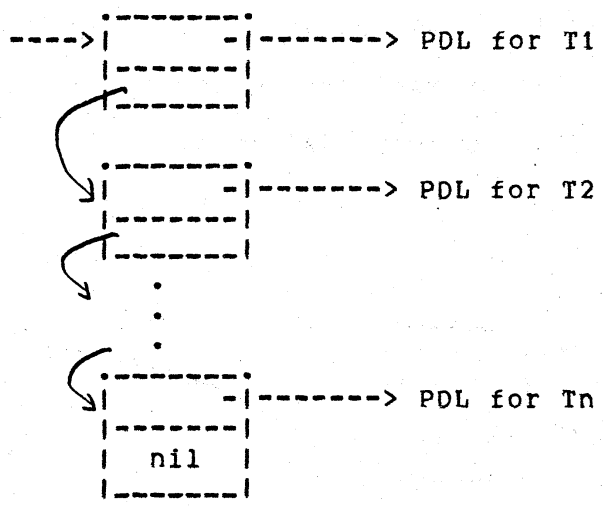
We call such a dymanic linkage done in exection time PDL linkage. Note that, when executing a non type-parameterized module, no PDL linkage is required. For each type-parameterized module M(P1:S1,...,Pn:Sn), PDL linkages are required when and only when some Pi ($1 \leq i \leq n$) is used as an actual type parameter to some formal type parameter of sype Si' such that $Si' \lesssim Si$ and that Si' differs from Si.

Type parameter lists

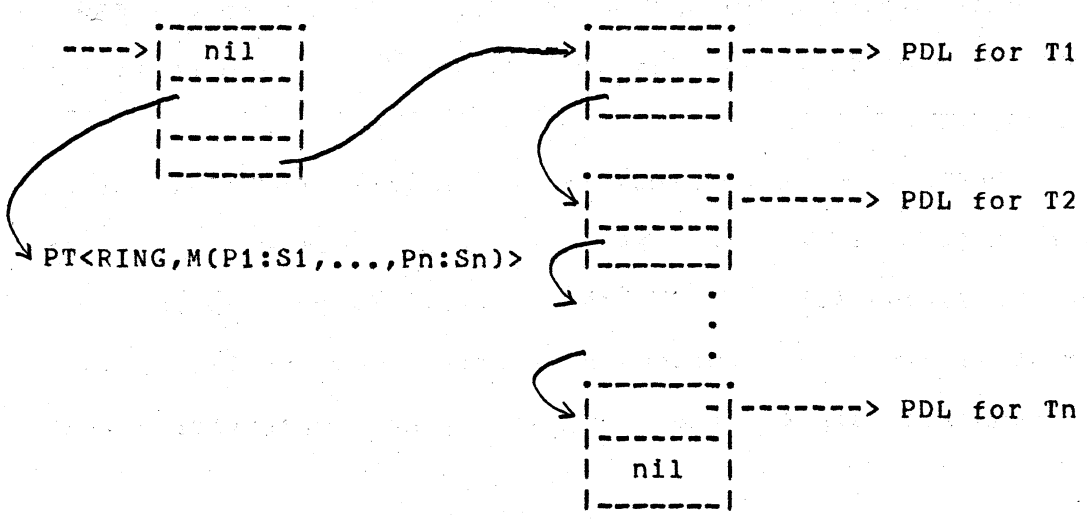
So far, we have considered only those cases where the actual type parameters to POLY(P:RING) are not type-parameterized. Now we explain how to deal with the cases where the actual type parameters to POLY(P:RING) are also type-parameterized.

Consider the case when POLY(M(T1,...,Tn))#ADD is called where; M(P1:S1,...,Pn:Sn) is a type-parameterized type

module with type parameters P1,...,Pn of sypes S1,...,Sn, respectively, and T1,...,Tn are actual type parameters to M(P1:S1,...,Pn:Sn). (T1,...,Tn may be themselves type-parameterized.) In this case, the PDL's for T1,...,Tn must be sent to POLY(P:RING). These PDL's are combined together in a list called a type parameter list (TPL) as shown below.

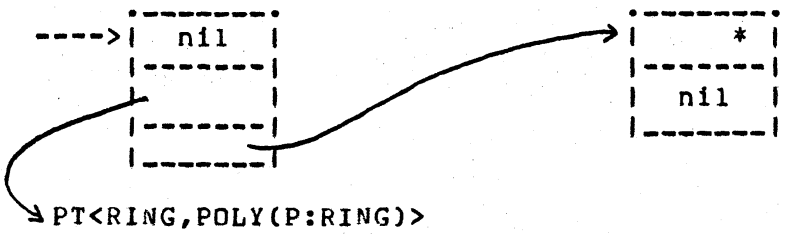



This TPL is linked from the PDL for M(P1:S1,...,Pn:Sn).



When M(P1:S1,...,Pn:Sn)#ADD is called in executing POLY(P:RING), each PDL for Ti is retrieved through the PDL for M(P1:S1,...,Pn:Sn) and sent to M(P1:S1,...,Pn:Sn)#ADD.

The TPL's must be constructed in execution time if a certain module N which calls POLY(M(T1,...,Tn))#ADD is a type-parameterized module and Ti coincides with one of the formal type parameters of N. For example, in the realization part of BIPOLY(P1:RING) (in Fig.1.2), POLY(POLY(P1))#ADD is called. In this case, the processor prepares an 'incomplete' TPL in compilation time, which is linked from the PDL for POLY(P:RING).




When BIPOLY(P1:RING)#ADD is called with some actual type parameter, say T, the cell marked '*' is linked to the PDL for T.

Such a process to construct a complete TPL in execution time is called a TPL linkage. Note that if POLY(POLY(P1))#ADD in the realization part of BIPOLY(P1:RING) is replaced by POLY(P1)#ADD, no TPL linkage is required since the actual type parameter that

BIPOLY(P1:RING) receives can be sent to POLY(P:RING)#ADD directly.

## 4. Runtime TPL/PDL linkages

The incomplete portions of TPL/PDL's (i.e. those which require dynamic linkage in execution time to construct information about actual type parameters) must be linked carefully so that the information already constructed is retained. When an incomplete TPL/PDL is linked in execution time, if the same TPL/PDL is already linked in order to construct information about actual type parameters of a currently active module instance, then this old information is violated. Such a situation may not occur so often in the actual programming. Theoretically, however, it is possible to create such a situation as shown in the following example:

Consider how BIPOLY(BIPOLY(INT))#ADD is executed (though this is quite a pathological case). Since POLY(POLY(P1))#ADD appears in the realization part of BIPOLY(P1:RING) (See Fig.2.1) and the actual type parameter to P1 is BIPOLY(INT), POLY(POLY(BIPOLY(INT)))#ADD will be called in executing BIPOLY(BIPOLY(INT))#ADD. Then the actual ADD for P#ADD in the realization part of POLY(P:RING) is POLY(BIPOLY(INT))#ADD, and so forth. The diagram below shows those functions which will be called in executing BIPOLY(BIPOLY(INT))#ADD, in order:

BIPOLY(BIPOLY(INT))#ADD

POLY(POLY(BIPOLY(INT)))#ADD

POLY(BIPOLY(INT))#ADD

BIPOLY(INT)#ADD

POLY(POLY(INT))#ADD

POLY(INT)#ADD

INT#ADD

Fig.4.1 shows the state of the runtime stack and PDL's when BIPOLY(BIPOLY(INT))#ADD is called. As mentioned in the previous section, the cell marked "*" must be linked to the PDL for the actual type parameter that BIPOLY(P1:RING) receives (to the PDL for BIPOLY(INT), in this case). Then, in the course of executing BIPOLY(BIPOLY(INT))#ADD, BIPOLY(INT)#ADD will be called. This time the same cell "*" is to be linked to the PDL for INT.

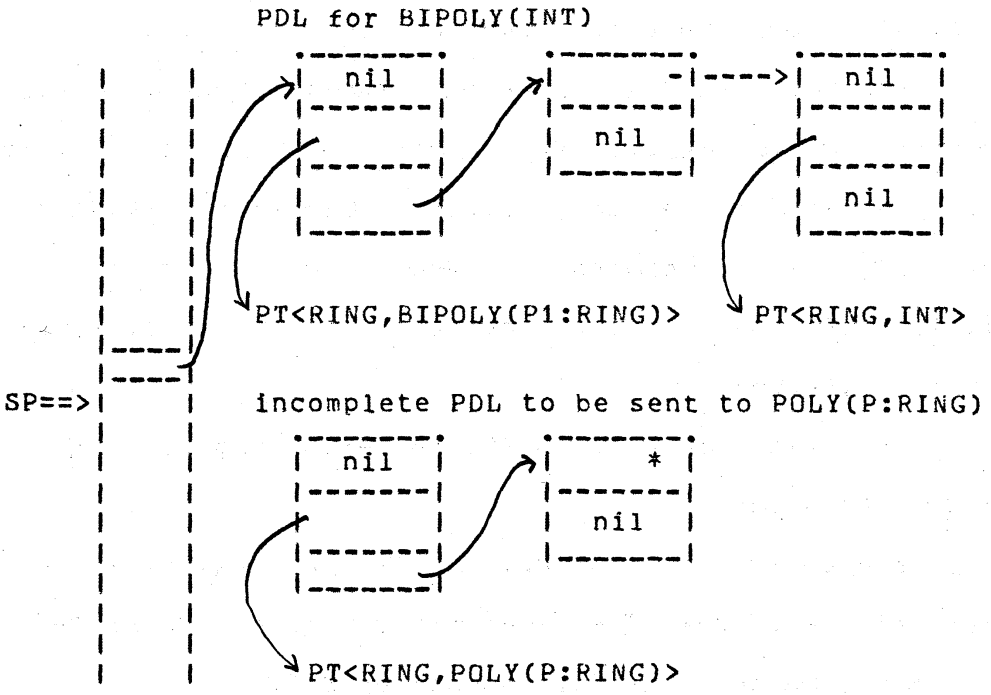



Fig.4.1 Just before BIPOLY(BIPOLY(INT))#ADD is called

Whenever such a violation of already constructed information occurs, the status of the TPL/PDL's must be restored when the function which causes the violation ends its execution. This situation arises for any type-parameterized module which requires dynamic TPL/PDL linkages in execution time and an instance of which is nested in another. Since the latter condition cannot be determined with the module-wise compilation, for any type module that requires dynamic TPL/PDL linkage, we must prepare for the situation above.

The well-known 'stack' mechanism is well adapted for the purpose. Before going to the actual mechanism adopted in the language processor, we present a virtual mechanism as an intermediate step.

Given a type-parameterized module M(P1:S1,...,Pn:Sn) which requires TPL/PDL linkages, suppose that the PDL's corresponding to $P_{\pi_1},\ldots,P_{\pi_m}$ must be linked, where $\{\pi_1,\ldots,\pi_m\}$ is a subset of $\{1,\ldots,n\}$. For each j ($1 \leq j \leq m$), a stack STj is prepared. When a function of M(P1:S1,...,Pn:Sn) is called from outside*** M(P1:S1,...,Pn:Sn),

1) the first node of each PDL corresponding to $P_{\pi_j}$ is pushed on STj, and

2) each cell in the incomplete TPL/PDL which must be linked to the actual type parameter corresponding to $P_{\pi_j}$ is set to point to the node just pushed on STj.

When the execution of a function of M(P1:S1,...,Pn:Sn)

which is called from outside*** M(P1:S1,...,Pn:Sn) is completed,

3) ST1,...,STm are popped, and

4) each cell in 2) is relinked to point to the node on the top of the stack.

Even if many module instances of a single module appear, the level of their nesting seems to stay low. Accordingly, these stacks need not be so large. This is the reason why we call these stacks small stacks. Thus the memory space is not wasted with this mechanism. Fig.4.2 shows two stages of the small stack for BIPOLY(P1:RING) in process of executing BIPOLY(BIPOLY(INT))#ADD.

---

***) When a function of M(P1:S1,...,Pn:Sn) is called from inside of the module, the actual type parameters remain unchanged. Thus the process is unnecessary.

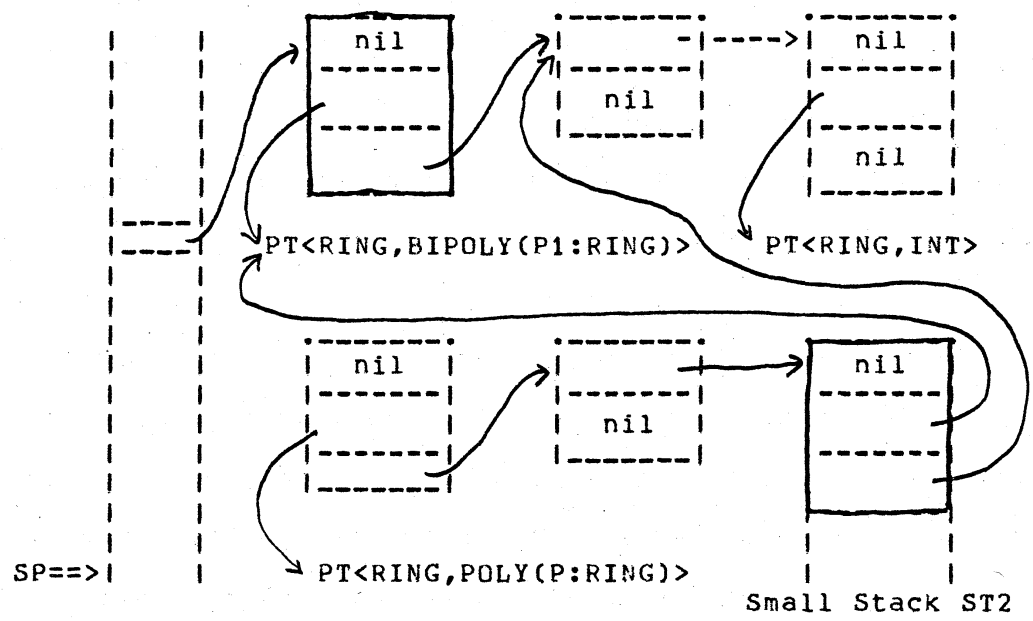



Fig.4.2(a) When BIPOLY(BIPOLY(INT))#ADD is called

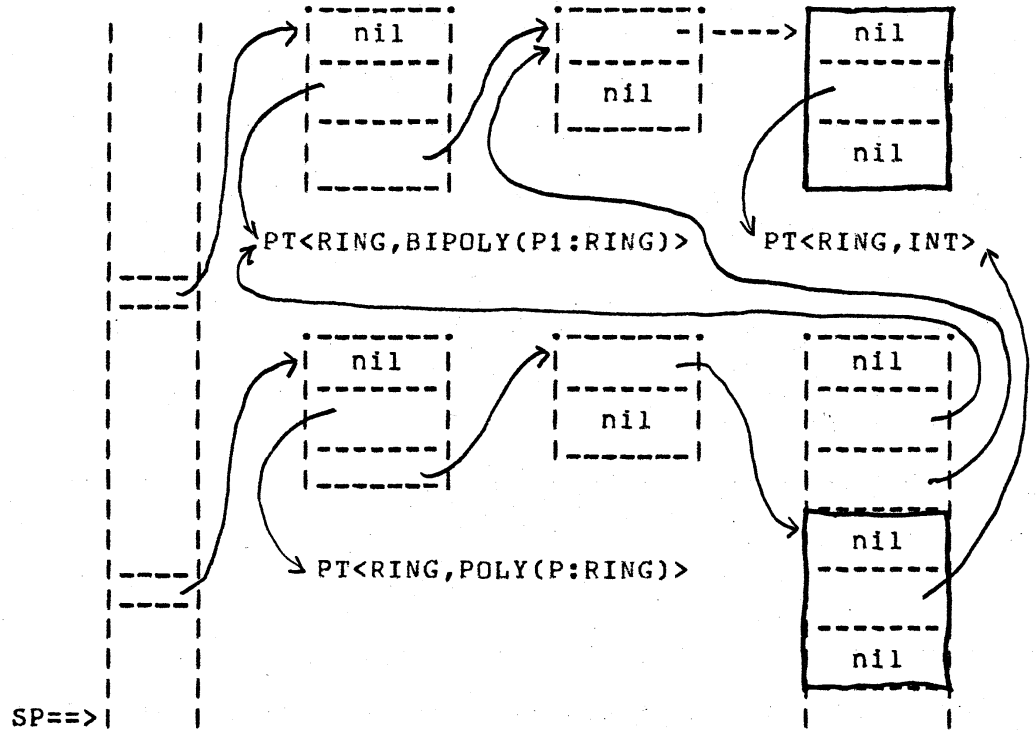



Fig.4.2(b) When BIPOLY(INT)#ADD is called

There is room for improvement to cover the following inefficiencies.

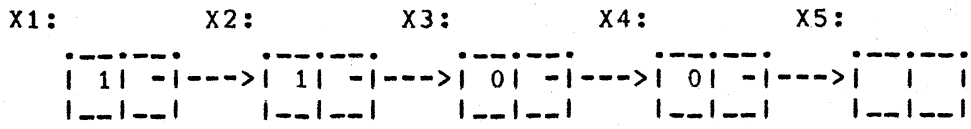
1. An entire node must be pushed on the small stack.
2. Each incomplete TPL/PDL must be linked and relinked. This may be a problem when a module requires many incomplete TPL/PDL's.

The improvement can be realized with the indirect addressing mechanism of DEC-20 which is also found in many other computer hardwares.

With this mechanism, one need only push on the small stacks the pointers to the PDL the module receives, not the entire node pointed to by the pointer. Moreover, TPL linkages are done automatically.

The DEC-20 CPU calculates effective address as follows (if no indexing is used): each memory and instruction word contains an 18-bit address part and a 1-bit indirect flag. If an instruction word must reference memory, its indirect flag is tested. If it is 0ff, the number in its address part is the effective address. If it is 00, addressing is indirect, and the processor retrieves another address word from the location specified by that address part. This new word is processed in exactly the same manner. This process continues until some referenced location is found with indirect flag 0ff: the number in its address part is the effective address.

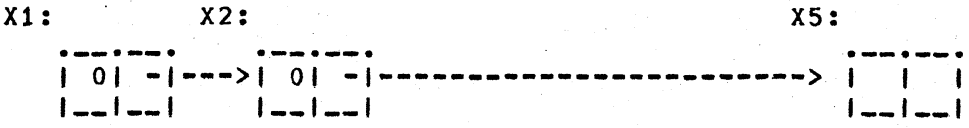
Suppose, for instance, that there is a chain of pointers as shown in the figure below.




Here each cell represents a word and the left hand side of each cell contains an indirect bit with 1 for on and 0 for off.

An indirect load instruction from X1 (i.e. indirect load instruction whose address part is X1) is executed as follows. The processor retrieves the content of X1. Since the indirect flag is on, it retrieves the content of X2. Again, the flag is on and the content of X3 is retrieved. The flag being off this time, X4 is the effective address. Thus the content of X4 (i.e. the pointer to X5) is loaded.

Note that the instruction yields the same result as above even when the chain of pointers are replaced as:




Now we are ready to explain the improved small stack mechanism.

As before, when M(P1:S1,...,Pn:Sn) is compiled, small stacks ST1,...,STm are prepared. In addtion, stack pointers SP1,...,SPm are prepared one for each STj. Each cell in the

incomplete TPL/PDL which must be linked to the actual type parameter corresponding to $P\pi_j$ contains the pointer to SPj and its indirect flag is set on. Each cell which contains the pointer to such a cell is also set the flag on. The flags of other cells are set off. (See Fig.4.3)

When a function of M(P1:S1,...,Pn:Sn) is called from outside of the module, each pointer to the PDL corresponding to $P\pi_j$ is pushed on STj. When the execution of a function of M(P1:S1,...,Pn:Sn) which is called from outside of the module is completed, each STj is simply popped.

With this method, Fig.4.2(a) is revised as follows.

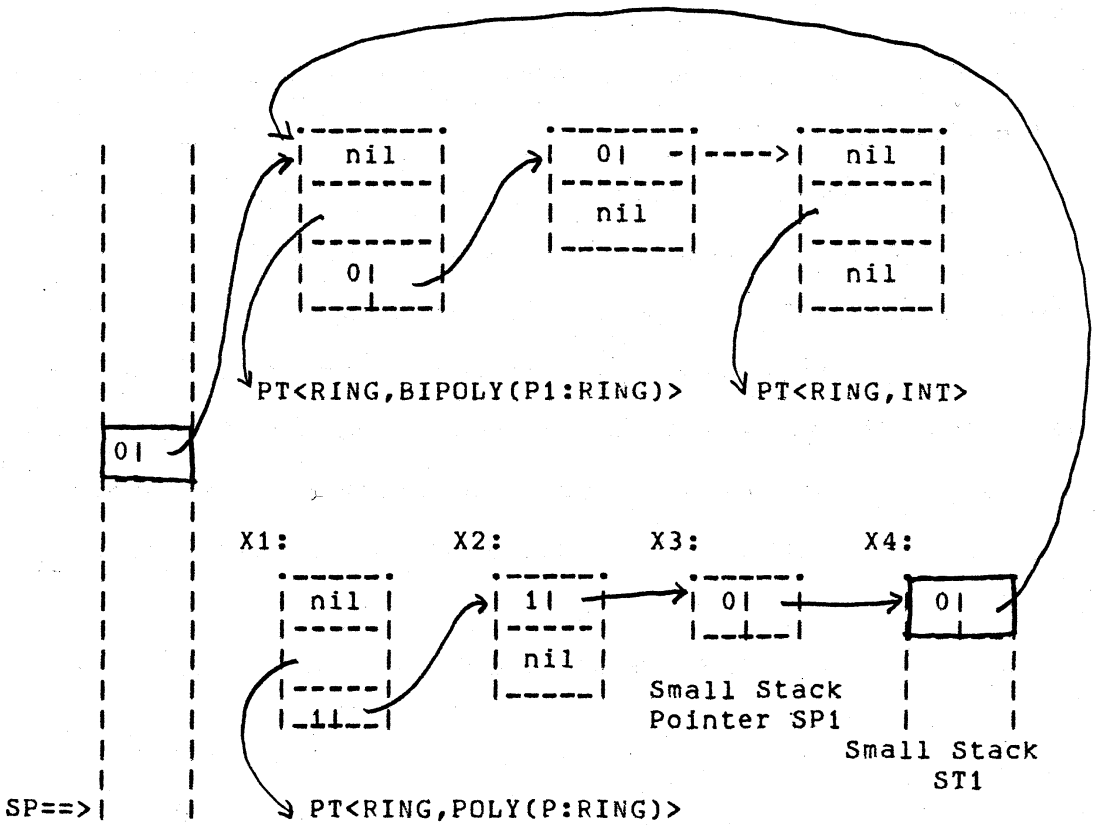



Fig.4.3 When BIPOLY(BIPOLY(INT))#ADD is called

Now return to the problem of how, in executing POLY(P:RING)#ADD, the actual ADD is retrieved and actual type parameters are sent to the module that the actual ADD belongs.

The P#ADD call in POLY(P:RING) is done as follows.

Step 1. Search the PDL it receives to find a node whose first cell is nil.

Step 2. Load the third word of the cell to a register, say, LX.

Step 3. If LX is not nil then load a pointer indirectly from LX and push it on the runtime stack. Else go to Step 5.

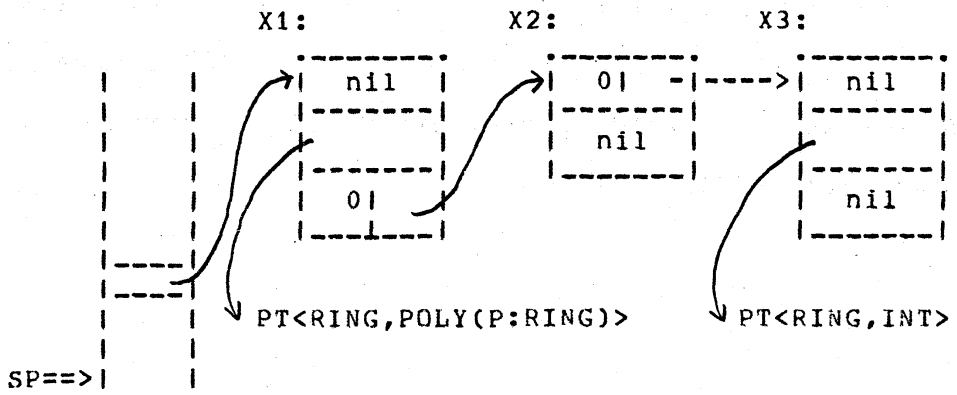
Step 4. Load the second word of the node pointed to by LX and go to Step 3.

Step 5. Get the actual ADD from the procedure table pointed to by the node found in Step 1 and call it.

(Step 1 and 5 are simplified for brevity.)

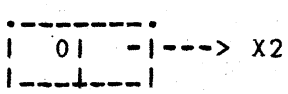
To see the soundness of the above algorithm, we trace the steps in two cases; One case when POLY(P:RING) receives a PDL generated in compile time and another case when the TPL is linked through the small stack in execution time.

Case 1. Suppose POLY(P:RING) receives a PDL of the form:




step 1. The search stops immediately since the first cell of node X1 is nil.

step 2. LX contains:

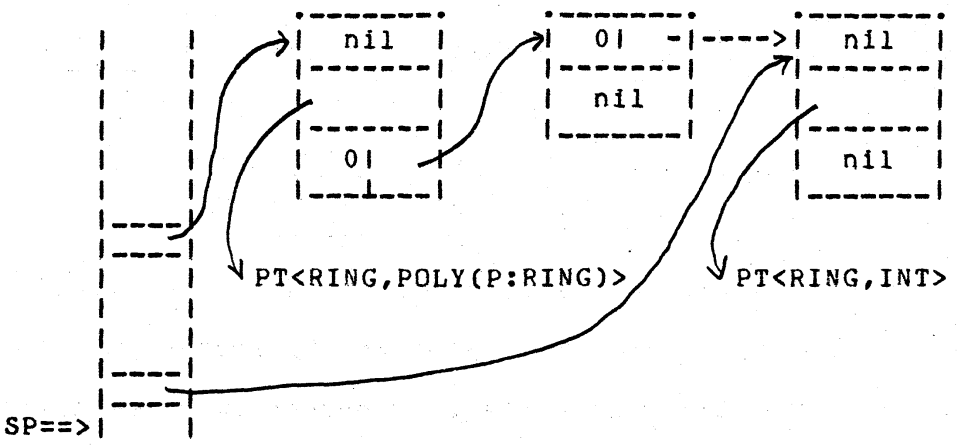



step 3. Since the indirect flag is off in LX, the effective address is X2. The first cell of X2 is pushed on the runtime stack.

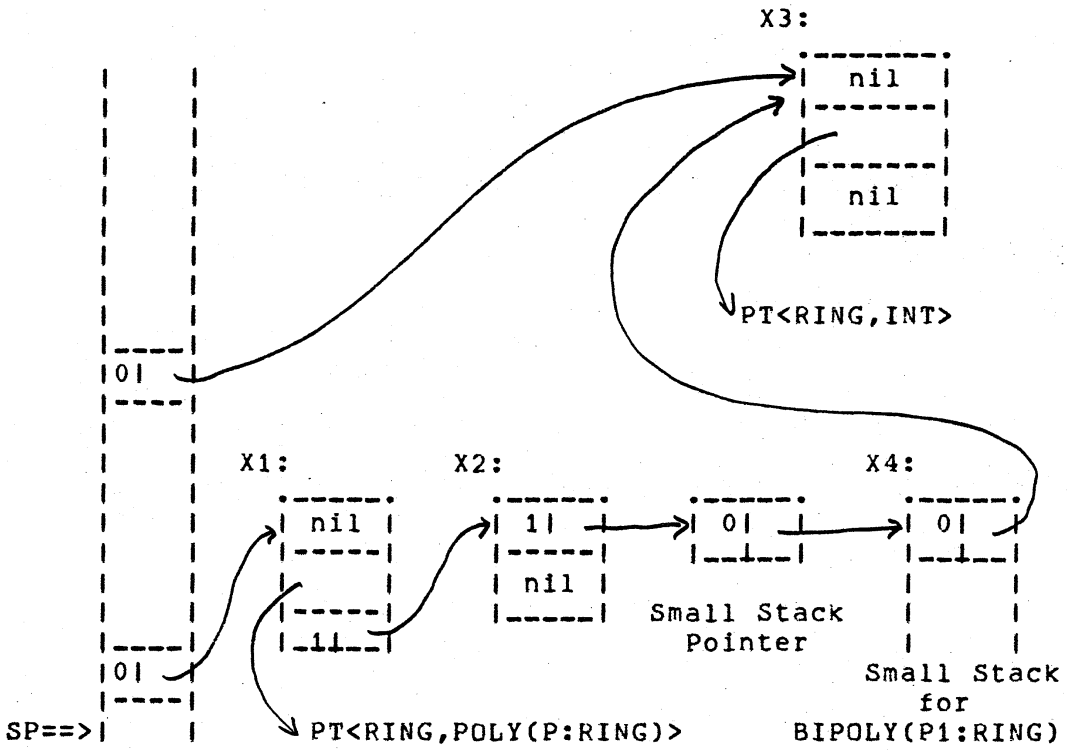
step 4. The second word of X2 is loaded to LX.

step 3. LX is nil.

step 5. POLY(P:RING)#ADD is called.

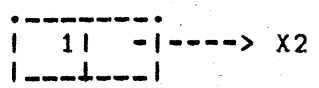


Case 2. Consider the execution of BIPOLY(INT)#ADD. When POLY(POLY(INT))#ADD is called from BIPLY(INT)#ADD, POLY(P:RING) receives a PDL, whose TPL is linked through the small stack for BIPOLY(P1:RING).




The trace is same as in case 1 except for the first Step 2 and Step 3.

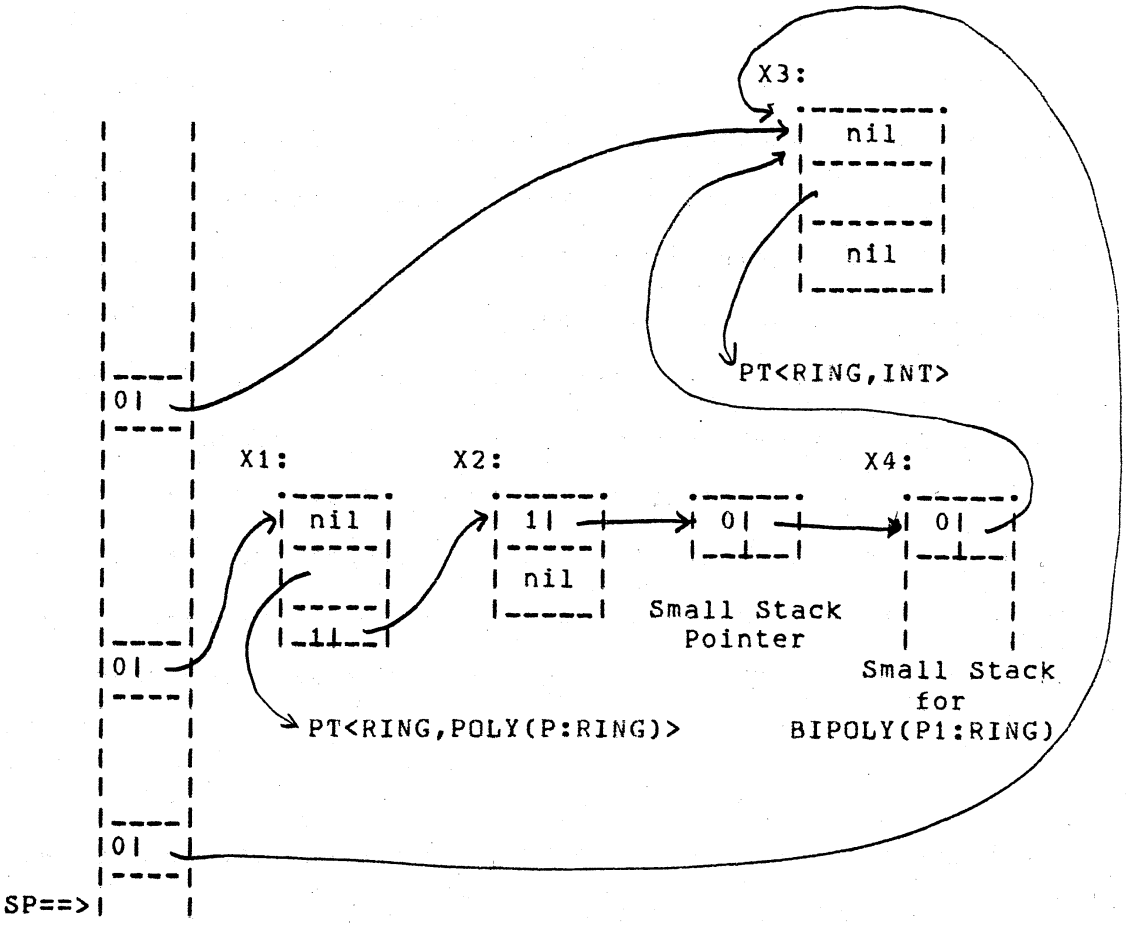
step 2. LX contains:




stpe 3. Since the indirect flag is on in LX, the effective address is X4. (See the example presented in the explanation of the indirect addressing mechanism.) The first

word of X4 (i.e. the pointer to X3) is pushed on the runtime stack.

When POLY(P:RING)#ADD is called at Step 5, the state of the runtime stack is:




Thus in both cases POLY(P:RING)#ADD is called with valid PDL set on the runtime stack.

In Step 1 of the algorithm, indirect instruction will be used as in Step 3 since the PDL may also be linked through small stacks. We leave it to the reader to detail Step 1 and 5 of the algorithm.

Note. Treatment of assignment and equality

As mentioned in section 1, every sype or type in ℓ is supposed to have its own EQUAL function. The truth value of the equality between two objects of a type T is determined by the EQUAL of T (i.e. T#EQUAL). If T#EQUAL is not defined in the realization part of the type module T, then the system automatically generates codes for T#EQUAL so that the EQUAL of the type by which T is represented is called.

The assignment (ASSIGN) is a data-type independent program construct in language ℓ and is never given implementation in the realization part of any type module. In the implementation of the language, however, it is convenient to consider that each type T has its own assignment among the basic operations of T and we conveniently denote it as 'T#ASSIGN' as if ASSIGN were a primitive function of T. Thus, for example, the assignment statement

X:=Y

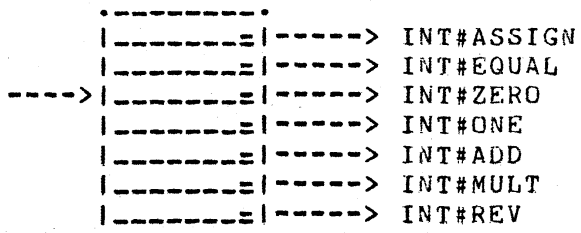
(where X and Y are variables of type T) is considered as:

T#ASSIGN(X,Y)

In this way, EQUAL's and ASSIGN's can be treated in the same manner as (other) primitive functions.

In executing POLY(P:RING), if P#EQUAL or P#ASSIGN is required, the actual EQUAL or the actual ASSIGN must also be retrieved from the procedure description list that POLY(P:RING) receives. Thus we extend each procedure table PT<S,T> so that its '-1'-th and '0'-th entries contain

T#ASSIGN and T#EQUAL, respectively. For example, PT<RING,INT> (in Fig.3.1) is extended as follows:




When the actual EQUAL or ASSIGN is retrieved, since they are always contained in the fixed entries in any procedure table, the intermediate adaptor tables in the PDL need not be used. Therefore the PDL is simply traversed to find the node which contains the procedure table. This indicates that the retrieval of the actual EQUAL or ASSIGN is faster than that of other primitive functions.

Remember that EQUAL is the only primitive function of sype ANY. For the same reason as above, for any sype or type S, we need no AT<ANY,S> at all. For example, when a function of ARRAY(P3:ANY) is called in POLY(P:RING), the PDL that POLY(P:RING) receives can be sent to ARRAY(P3:ANY) as it is, without TPL linkage. Actually, most of the sype-sype or sype-type relations are of the form ANY$\lesssim$S. So this consideration may greatly increase the efficiency.

ACKNOWLEDGEMENTS

The author wishes to express his appreciation to Professor Reiji Nakajima for patiently supervising this research.

REFERENCES

1. Burstall, R., Goguen, J.: Putting theories together to make specifications. Int. Joint Conf. on A.I., 1977

2. Liskov, B. et al.: Abstraction mechanisms in CLU. Comm. ACM. 8. 567-576 1977

3. Honda, M., Nakajima, R.: Interactive theorem proving on Hierarchically and Modularly Structured Set of Very Many Axioms. 6th Int. Joint Conf. on Artificial Intelligence 79

4. Nakajima, R., Honda, M., Nakahara, H.: Describing and verifying programs with abstract data types. Formal description of Programming Concepts. (ed. Neuhold) North-Holland Publishing. Co. 1977

5. Nakajima, R., Nakahara, H., Honda, M.: Hierarchical Program Specification and Verification--a Many-Sorted Logical Approach-- RIMS-Preprint 265 1978

6. Nakajima, R., Yuasa, T.: Program Development with the ζ system. Proc. of TSC Symposium on Intelligent Programming Systems, IBM Japan 1978

7. Nakajima, R.: Sypes - partial types - for program structuring and first order system ζ logic. Research Report No.22, Institute of Informatics, Univ. of Oslo 1977

8. Yuasa, T.: Supports for Hierarchical Software Development --Systems and Mathematical methods-- (Master's Thesis)